BISM: Bytecode-Level Instrumentation for Software Monitoring

Chukri Soueidi, Ali Kassem, Yliès Falcone

BISM: Bytecode-Level Instrumentation for Software Monitoring

Chukri Soueidi, Ali Kassem, and Yliès Falcone
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
chukri.a.soueidi@inria.fr; ali.kassem@inria.fr; ylies.falcone@inria.fr

Abstract. BISM (Bytecode-Level Instrumentation for Software Monitoring) is a lightweight Java bytecode instrumentation tool which features an expressive high-level control-flow-aware instrumentation language. The language follows the aspect-oriented programming paradigm by adopting the joinpoint model, advice inlining, and separate instrumentation mechanisms. BISM provides joinpoints ranging from bytecode instruction to method execution, access to comprehensive context information, and instrumentation methods. BISM runs in two modes: build-time and load-time. We demonstrate BISM effectiveness using two experiments: a security scenario and a general runtime verification case. The results show that BISM instrumentation incurs low runtime and memory overheads.

Keywords: Instrumentation · Runtime Verification · Monitoring · Java Bytecode · Aspect-Oriented Programming · Control Flow · Static and Dynamic Contexts.

1 Introduction

Instrumentation is essential to the software monitoring workflow [9,3]. Instrumentation allows extracting information from a running software to abstract the execution into a trace that is fed to a monitor. Depending on the information needed by the monitor, the granularity level of the extracted information may range from coarse (e.g., a function call) to fine (e.g., an assignment to a local variable, a jump in the control flow).

Aspect-oriented programming (AOP) [14] is a popular and convenient paradigm where instrumentation is a cross-cutting concern. For Java programs, runtime verification tools [10,2] have for long relied on AspectJ [13], which is one of the reference AOP implementations for Java. AspectJ provides a high-level pointcut/advice model for convenient instrumentation. However, AspectJ does not offer enough flexibility to perform some instrumentation tasks that require to reach low-level code regions, such as bytecode instructions, local variables of a method, and basic blocks in the control-flow graph (CFG).

Yet, there are several low-level Java bytecode manipulation frameworks such as ASM [6] and BCEL [19]. However, instrumenting programs with such frameworks is tedious and requires expertise on the bytecode. Other Java bytecode instrumentation frameworks, from which DiSL [16] is the most remarkable, enable flexible low-level instrumentation and, at the same time, provide a high-level language. However, DiSL does not allow inserting bytecode instructions directly but provides custom transformers where a developer needs to revert to low-level bytecode manipulation frameworks. This makes various scenarios tedious to implement in DiSL and often at the price of a considerable bytecode overhead.

Contributions. In this paper, we introduce BISM (Bytecode-Level Instrumentation for Software Monitoring), a lightweight Java bytecode instrumentation tool that features an expressive high-level instrumentation language. The language inspires from the AOP paradigm by adopting the joinpoint model, advice inlining, and separate instrumentation mechanisms. In particular, BISM provides a separate Java class to specify instrumentation code, and offers a variety of *joinpoints* ranging from bytecode instruction to basic block and method execution. BISM also provides access to a set of comprehensive joinpoint-related *static* and *dynamic contexts* to retrieve some relevant information, and a set of *instrumentation methods* to be called at joinpoints to insert code, invoke methods, and print information. BISM is control-flow aware. That is, it generates CFGs for all methods and offers this information at joinpoints and context objects. Moreover, BISM provides a variety of control-flow properties, such as capturing conditional jump branches and retrieving successor and the predecessor basic blocks. Such features provide support to future tools using a control-flow analysis, for instance, in the security domain, to detect control-flow attacks, such as test inversions and arbitrary jumps.

We demonstrate BISM effectiveness using two complementary experiments. The first experiment shows how BISM can be used to instrument a program to detect test inversion attacks. For this purpose, we use BISM to instrument AES (Advanced Encryption Standard). The second experiment demonstrates a general runtime verification case where we use BISM to instrument seven applications from the DaCapo benchmark [5] to verify the classical **HasNext**, **UnsafeIterator** and **SafeSyncMap** properties. We compare the performance of BISM, DiSL, and AspectJ in build-time and load-time instrumentation, using three metrics: size, memory footprint, and execution time. In build-time instrumentation, the results show that the instrumented code produced by BISM is smaller, incurs less overhead, and its execution incurs less memory footprint. In load-time instrumentation, the load-time weaving and the execution of the instrumented code are faster with BISM.

Paper organization. Section 2 overviews the design goals and the features of BISM. Section 3 introduces the language featured by BISM. Section 4 presents the implementation of BISM. Section 5 reports on the case studies and a comparison between BISM, DiSL, and AspectJ. Section 6 discusses related work. Section 7 draws conclusions.

2 BISM Design and Features

BISM is implemented on top of ASM [6], with the following goals and features.

Instrumentation mechanism. BISM language follows the AOP paradigm. It provides a mechanism to write separate instrumentation classes. An instrumentation class specifies the instrumentation code to be inserted in the target program at chosen joinpoints. BISM offers joinpoints that range from bytecode instruction to basic block and method execution. It also offers several instrumentation methods and, additionally, accepts instrumentation code written in the ASM syntax. The instrumentation code is eventually compiled by BISM into bytecode instructions and inlined in the target program.

Access to program context. BISM offers access to complete static information about instructions, basic blocks, methods, and classes. It also offers dynamic context objects that provide access to values that will only be available at runtime such as values of local

variables, stack values, method arguments, and results. Moreover, BISM allows accessing instance and static fields of these objects. Furthermore, new local variables can be created within the scope of a method to (for instance) pass values between joinpoints.

Control flow context. BISM generates the CFGs of target methods out-of-the-box and offers this information within joinpoints and context objects. In addition to basic block entry and exit joinpoints, BISM provides specific control-flow related joinpoints such as `OnTrueBranchEnter` and `OnFalseBranchEnter`, which capture conditional jump branches. Moreover, it provides a variety of control-flow properties within the static context objects. For example, it is possible to traverse the CFG of a method to retrieve the successors and the predecessors of basic blocks. Furthermore, BISM provides an optional feature to display the CFGs of methods before and after instrumentation.

Compatibility with ASM. BISM uses ASM extensively and relays all its generated class representations within the static context objects. Furthermore, it allows for inserting raw bytecode instructions by using the ASM data types. In this case, it is the responsibility of the user to write instrumentation code free from compilation and runtime errors. If the user unintentionally inserts faulty instructions, the code might break. The ability to insert ASM instructions provides highly expressive instrumentation capabilities, especially when it comes to inlining the monitor code into the target program.

Bytecode coverage. BISM can run in two modes: *build-time* (as a standalone application) with static instrumentation, and *load-time* with an agent (utilizing `java.lang.instrument`) that intercepts all classes loaded by the JVM and instruments before the linking phase. In build-time, BISM is capable of instrumenting all the compiled classes and methods[1]. In load-time, BISM is capable of instrumenting additional classes, including classes from the Java class library that are flagged as modifiable. The modifiable flag keeps certain core classes outside the scope of BISM. Note, modifying such classes is rather needed in dynamic program analysis (e.g., profiling, debugging).

3 BISM Language

We demonstrate the language in BISM, which allows developers to write *transformers* (*i.e.,* instrumentation classes). The language provides joinpoints which capture exact execution points, static and dynamic contexts which retrieve relevant information at joinpoints, and instrumentation methods used to instrument a target program.

Joinpoints. Joinpoints identify specific bytecode locations in the target program. BISM offers joinpoints that capture bytecode instruction executions: `BeforeInstruction` and `AfterInstruction`, conditional jump branches: `OnTrueBranchEnter` and `OnFalseBranchEnter`, executions of basic blocks: `OnBasicBlockEnter` and `OnBasicBlockExit`, method executions: `OnMethodEnter` and `OnMethodExit`, and method calls: `BeforeMethodCall` and `AfterMethodCall`.

[1] Excluding the native and abstract methods, as they do not have bytecode representation.

```
public class BasicBlockTransformer extends Transformer {
  @Override
  public void onBasicBlockEnter(BasicBlock bb){
    String id = bb.method.className+"."+bb.method.name+"."+bb.id;
    print("Entered block:" + id); }
  @Override
  public void onBasicBlockExit(BasicBlock bb){
    String id = bb.method.className+"."+bb.method.name+"."+bb.id;
    print("Exited block:" + id); }
}
```

Listing 1.1: A transformer for intercepting basic block executions.

Static context. Static context objects provide relevant static information at joinpoints. These objects can be used to retrieve information about a bytecode instruction, a method call, a basic block, a method, and a class. BISM performs static analysis on target programs and provides additional control-flow-related static information such as basic block successors and predecessors. Listing 1.1 shows a transformer using joinpoints `onBasicBlockEnter` and `onBasicBlockExit` to intercept all basic block executions. The static context `BasicBlock bb` is used to get the block id, the method name, and the class name. Here, the instrumentation method `print` inserts a print invocation in the target program before and after every basic block execution.

```
public class IteratorTransformer extends Transformer {
 @Override
 public void afterMethodCall(MethodCall mc,
     MethodCallDynamicContext dc){
   if (mc.methodName.equals("iterator") &&
       mc.methodOwner.endsWith("List")) {
     DynamicValue callingClass = dc.getThis(mc); // Access to
     DynamicValue list = dc.getMethodReceiver(mc); // dynamic
     DynamicValue iterator = dc.getMethodResult(mc); // data
     StaticInvocation sti = // Instrumenting to invoke a monitor
       new StaticInvocation("IteratorMonitor","iteratorCreation");
     sti.addParameter(callingClass);
     sti.addParameter(list);
     sti.addParameter(iterator);
     invoke(sti); }
 }}
```

Listing 1.2: A transformer that intercepts the creation of an iterator from a `List`.

Dynamic context. Dynamic Context objects provide access to dynamic values that are possibly only known during execution. BISM gathers this information from local variables and operand stack, then weaves the necessary code to extract this information. In some cases (e.g., when accessing stack values), BISM might instrument additional local variables to store them for later use. We list the methods available in dynamic contexts: `getThis`, `getLocalVariable`, `getStackValue`, `getInstanceField` and `getStaticField`, and the values related to these methods: `getMethodReceiver`, `getMethodArgs`, and `getMethodResult`. BISM also allows inserting and up-

dating new local variables within the scope of a method. Listing 1.2 presents a transformer using `afterMethodCall` joinpoint to capture the return of an `Iterator` created from a `List` object, and retrieving dynamic data from the dynamic context object `MethodCallDynamicContext dc`. The example also shows how to limit the scope using an if-statement to a specific method. Note that BISM also provides a general notion of *scope* that can be specified as an argument to match packages, classes, and methods by names (using possibly wildcards).

Instrumentation methods. A developer instruments the target program using specified instrumentation methods. BISM provides `print` methods with multiple options to invoke a print command. It also provides (i) `invoke` methods for static method invocation and (ii) `insert` methods for bytecode instruction insertion. These methods are compiled by BISM into bytecode instructions and inlined at the exact joinpoint locations. Listing 1.1 shows the use of `print` to print the constructed id of a basic block. Listing 1.2 shows how a method invocation is instrumented after a method call.

4 BISM Implementation

BISM is implemented in Java with about 4,000 LOC and 40 classes distributed in separate modules [17]. It uses ASM for bytecode parsing, analysis, and weaving. Figure 1 shows BISM internal workflow.

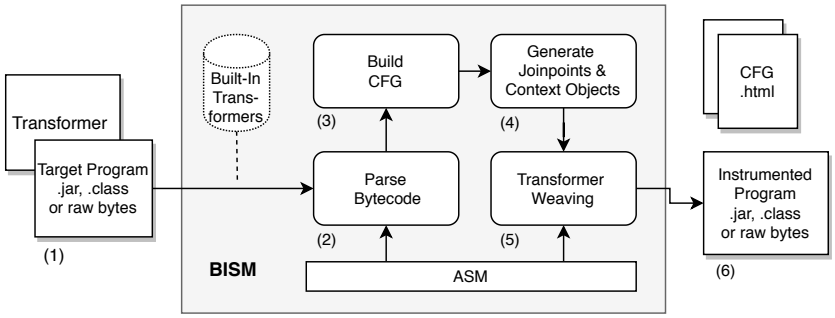


Fig. 1: Instrumentation process in BISM.

(1) User Input. In build-time mode, BISM takes a target program bytecode (*.class* or *.jar*) to be instrumented, and a transformer which specifies the instrumentation logic. In load-time mode, BISM only takes a transformer used to instrument every class being loaded by the JVM. BISM provides several built-in transformers that can be directly used. Moreover, users can specify a scope to filter target packages, classes, or methods.
(2) Parse Bytecode. BISM uses ASM to parse the bytecode and to generate a tree object which contains all the class details, such as fields, methods, and instructions.
(3) Build CFG. BISM constructs the CFGs for all methods in the target class. If the specified transformer utilizes control-flow joinpoints (i.e., `onTrueBranch` and `onFalseBranch`), BISM eliminates all *critical edges* from the CFGs to avoid instrumentation errors. This is done by inserting empty basic blocks in the middle of critical edges. Note, BISM keeps copies of the original CFGs. Users can optionally enable the *visualizer* to store CFGs in HTML files on the disk.

(4) Generate Joinpoints and Context Objects. BISM iterates over the target class to generate all joinpoints utilizing the created CFGs. At each joinpoint, the relevant static and dynamic context objects are created.
(5) Transformer Weaving. BISM evaluates the used dynamic contexts based on the joinpoint static information and weaves the bytecode needed to extract concrete values from executions. It then weaves instrumentation methods by compiling them into bytecode instructions that are woven into the target program at the specified joinpoint.
(6) Output. The instrumented bytecode is then output back as a *.class* file in build-time mode, or passed as raw bytes to the JVM in load-time mode. In case of instrumentation errors, e.g., due to adding manual ASM instructions, BISM emits a weaving error. If the visualizer is enabled, instrumented CFGs are stored in HTML files on the disk.

5 Evaluation

We compare BISM with DiSL and AspectJ using two complementary experiments. To guarantee fairness, we switched off adding exception handlers around instrumented code in DiSL. In what follows, we illustrate how we carried out our experiments and the obtained results[2].

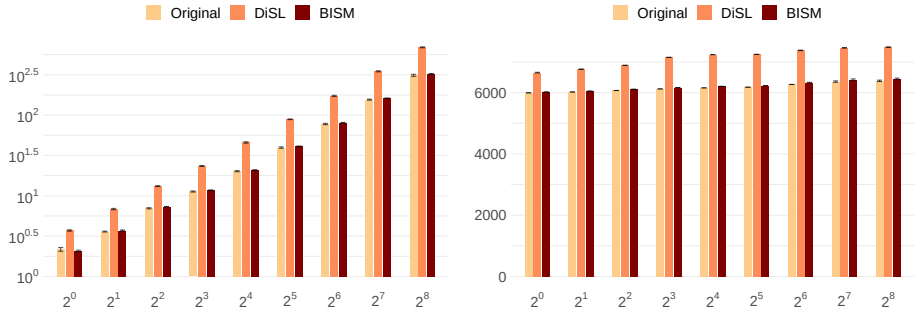
5.1 Inline Monitor to Detect Test Inversions

We instrument an external AES (Advanced Encryption Standard) implementation in build-time mode to detect test inversions. The instrumentation deploys inline monitors that duplicate all conditional jumps in their successor blocks to report test inversions. We implement the instrumentation as follows:

- In BISM, we use built-in features to duplicate conditional jumps utilizing `insert` instrumentation method to add raw bytecode instructions. In particular, we use the `beforeInstruction` joinpoint to capture all conditional jumps. We extract the opcode from the static context object `Instruction` and we use the instrumentation method `insert` to duplicate the needed stack values. We then use the control-flow joinpoints `OnTrueBranchEnter` and `onFalseBranchEnter` to capture the blocks executing after the jump. Finally, at the beginning of these blocks, we utilize `insert` to duplicate conditional jumps.
- In DiSL, we implement a custom `InstructionStaticContext` object to retrieve information from conditional jump instructions such as the index of a jump target and instruction opcode. Note, we use multiple `BytecodeMarker` snippets to capture all conditional jumps. To retrieve stack values, we use the dynamic context object. Finally, on successor blocks, we map opcodes to Java syntax to re-evaluate conditional jumps using switch statements.

We use AES to encrypt and then decrypt input files of different sizes, line by line. The bytecode size of the original AES class is 9 KB. After instrumentation, it is 10 KB (+11.11%) for BISM, and 128 KB (+1,322%) for DiSL. The significant overhead in DiSL is due to the inability to inline the monitor in bytecode and having to instrument it in Java. We note that it is not straightforward in DiSL to extract control-flow information

[2] More details about the experiments are at https://gitlab.inria.fr/monitoring/bism-experiments.



(a) Runtime (ms). (b) Memory footprint (KB).

Fig. 2: Runtime and memory footprint by AES on files of different sizes.

in Markers, whereas BISM provides this out-of-the-box. Figure 2 reports runtime and memory footprint with respect to file size (KB)[3]. For each input file, we performed 100 measurements and reported the mean and the standard deviation. The latter is very low. We use Java JDK 8u181 with 4 GB maximum heap size on a standard PC (Intel Core i7 2.2 GHz, 16 GB RAM) running macOS Catalina v10.15.5 64-bit. The results show that BISM incurs less overhead than DiSL for all file sizes. Table 1 reports the number of events (corresponding to conditional jumps).

Table 1: Number of events according to the file input to AES (in millions).

Input File (KB)	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
Events (M)	0.92	1.82	3.65	7.34	14.94	29.53	58.50	117.24	233.10

5.2 DaCapo Benchmarks

Experimental setup. We compare BISM, DiSL, and AspectJ in a general runtime verification scenario[4]. We instrument the benchmarks in the DaCapo suite [5] (dacapo-9.12-bach), to monitor the classical **HasNext**, **UnSafeIterator**, and **SafeSyncMap** properties[5]. We only target the packages specific to each benchmark and do not limit our scope to `java.util` types; instead, we match freely by type and method name. We implement an external monitor library with stub methods that only count the number of received events.

We implement the instrumentation as follows:

- In BISM, we use the static context provided at method call joinpoints to filter methods by their names and owners. To access the method calls' receivers and results, we utilize the methods available in dynamic contexts.
- In DiSL, we implement custom Markers to capture the needed method calls and use argument processors and dynamic context objects to access dynamic values.

[3] Note, AspectJ is not suited for inline monitoring, and that is why it is not included.

[4] We use the latest DiSL version from https://gitlab.ow2.org/disl/disl and AspectJ Weaver 1.9.4.

[5] **HasNext** property specifies that a program should always call `hasNext()` before calling `next()` on an iterator. **UnSafeIterator** property specifies that a collection should not be updated when an iterator associated with it is being used. **SafeSyncMap** property specifies that a map should not be updated when an iterator associated with it is being used.

– In AspectJ, we use the call pointcut, type pattern matching and joinpoint static information to capture method calls and write custom advices that invoke the monitor.

We use Java JDK 8u251 with 2 GB maximum heap size on an Intel Core i9-9980HK (2.4 GHz. 8 GB RAM) running Ubuntu 20.04 LTS 64-bit. All our measurements correspond to the mean of 100 runs on each benchmark, calculating the standard deviation. We run our experiment in two modes: load-time and build-time. The first mode is to compare the performance of the tools in load-time instrumentation and the second mode to examine the performance of the generated instrumentation bytecode.

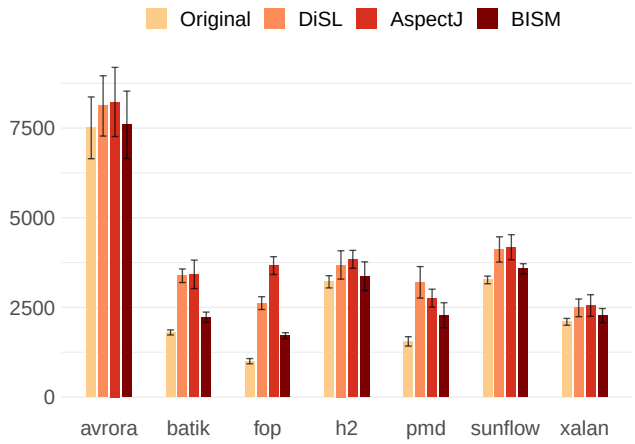


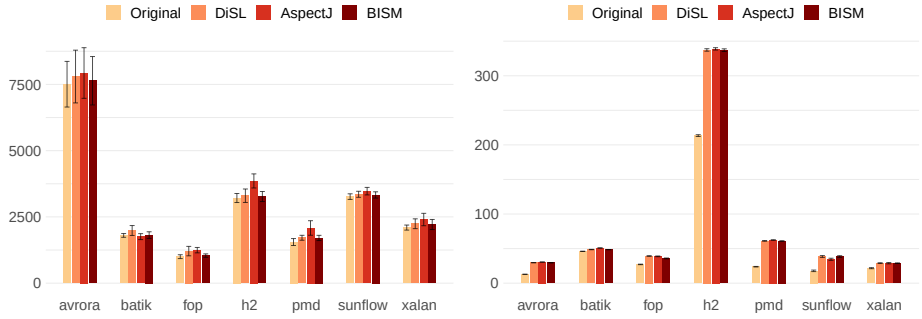
Fig. 3: Load-time instrumentation runtime (ms).

Load-time evaluation. Figure 3 reports the execution time in ms for the benchmarks. We do not measure the used memory since DiSL performs instrumentation on a separate JVM process. BISM shows better performance over DiSL and AspectJ in all benchmarks. DiSL shows better performance than AspectJ except for the pmd benchmark. For the pmd benchmark, this is mainly due to the fewer events emitted by AspectJ (see Table 2). We notice that AspectJ captures fewer events in benchmarks batik, fop, pmd, and sunflow. This is due to its inability to instrument synthetic bridge methods generated by the compiler after type erasure in generic types.

Build-time evaluation. We replace the original classes in the benchmarks with statically instrumented classes from each tool. Figure 4 reports the execution time and memory footprint of the benchmarks. For memory, we measure the used heap and non-heap memory after a forced garbage collection at the end of each run[6]. BISM shows less overhead in all benchmarks in execution time, except for batik where AspectJ emits fewer events. BISM also shows less overhead in used-memory footprint, except for sunflow, where AspectJ emits much fewer events.

Table 2 compares the instrumented bytecode and the number of events emitted after running the code. We report the number of classes in scope (Scope) and the instrumented (Instr.), we measure the overhead percentage (Ovh.) on the bytecode size for

[6] The DaCapo callback mechanism captures the end of each run.



(a) Runtime (ms). (b) Memory footprint (MB).

Fig. 4: Build-time execution.

each tool. We also report the number of generated events. BISM and DiSL emit the same number of events, while Aspect (AJ) produces fewer events due to the reasons mentioned above. The results show that BISM incurs less bytecode size overhead for all benchmarks. We notice that even with exception-handlers turned off, DiSL still wraps a targeted region with try-finally blocks when the `@After` annotation is used. This guarantees that an event is emitted after a method call, even if an exception is thrown.

Table 2: Generated bytecode size and events emitted.

	Scope	Instr.	Ref	BISM		DiSL		AspectJ		Events (M)	
			KB	KB	Ovh.%	KB	Ovh.%	KB	Ovh.%	#	AJ
avrora	1,550	35	257	264	2.72	270	5.06	345	34.24	2.5	2.5
batik	2,689	136	1,544	1,572	1.81	1,588	2.85	1,692	9.59	0.5	0.4
fop	1,336	172	1,784	1,808	1.35	1,876	5.16	2,267	27.07	1.6	1.5
h2	472	61	694	704	1.44	720	3.75	956	37.75	28	28
pmd	721	90	756	774	2.38	794	5.03	980	29.63	6.6	6.3
sunflow	221	8	69	71	2.90	74	7.25	85	23.19	3.9	2.6
xalan	661	9	100	101	1.00	103	3.00	116	16.00	1	1

6 Related Work and Discussion

Low-level code instrumentation is widely used for monitoring software and implementing dynamic analysis tools. To this end, several tools and frameworks, in different programming languages, have been proposed and adopted. We focus our comparison on Java-related instrumentation tools. Yet, there are several tools to instrument programs in different programing languages. For instance, to instrument C/C++ programs AspectC/C++ [7,18] (high-level) and LLVM [15] (low-level) are widely used.

ASM [6] is a bytecode manipulation framework utilized by several tools, including BISM. ASM offers two APIs that can be used interchangeably to parse, load, and modify classes. However, to use ASM, a developer has to deal with the low-level details of bytecode instructions and the JVM. BISM offers extended ASM compatibility and provides abstraction with its aspect-oriented paradigm.

DiSL is a bytecode-level instrumentation framework designed for dynamic program analysis [16]. DiSL adopts an aspect-oriented paradigm. It provides an extensible join-point model and access to static and dynamic context information. Even though BISM

provides a fixed set of joinpoints and static context objects, it performs static analysis on target programs to offer out-of-the-box additional and needed control-flow joinpoints with full static information. As for dynamic context objects, both BISM and DiSL provide equal access. However, DiSL provides typed dynamic objects. Also, both are capable of inserting synthetic local variables (restricted to primitive types in BISM). Both BISM and DiSL require basic knowledge about bytecode semantics from their users. In DiSL, writing custom markers and context objects also requires additional ASM syntax knowledge. However, DiSL does not allow the insertion of arbitrary bytecode instructions but provides a mechanism to write custom transformers in ASM that runs before instrumentation. Whereas, BISM allows to directly insert bytecode instructions, as seen in Sec. 5.1. Such a mechanism is essential in many runtime verification scenarios. All in all, DiSL provides more features (mostly targeted for writing dynamic analysis tools) and enables dynamic dispatch amongst multiple instrumentations and analysis without interference [4], while BISM is more lightweight as shown by our evaluation.

AspectJ [13] is the standard aspect-oriented programming [14] framework highly adopted for instrumenting Java applications. It provides a high-level language used in several domains like monitoring, debugging, and logging. AspectJ cannot capture bytecode instructions and basic blocks directly, forcing developers to insert additional code (like method calls) to the source program. With BISM, developers can target single bytecode instructions and basic block levels, and also have access to local variables and stack values. Furthermore, AspectJ introduces a significant instrumentation overhead, as seen in Sec. 5.2, and provides less control on where instrumentation snippets get inlined. In BISM, the instrumentation methods are weaved with minimal bytecode instructions and are always inlined next to the targeted regions.

7 Conclusions

BISM is an effective tool for low-level and control-flow aware instrumentation, complementary to DiSL, which is better suited for dynamic analysis (e.g., profiling). Our first evaluation (Sec. 5.1) let us observe a significant advantage of BISM over DiSL due to BISM's ability to insert bytecode instructions directly, hence optimizing the instrumentation. Our second evaluation (Sec. 5.2) confirms that BISM is a lightweight tool that can be used generally and efficiently in runtime verification. We notice a similar bytecode performance between BISM and DiSL after static instrumentation since, in both tools, the instrumentation (monitor invocation) is always inlined. On the other hand, AspectJ instruments calls to advice methods that, in turn, invoke the monitors. In load-time instrumentation, the gap between BISM and DiSL is smaller in benchmarks with a large number of classes in scope and a small number of instrumented classes. This stems from the fact that BISM performs a full analysis of the classes in scope to generate its static context. While DiSL generates static context only after marking the needed regions, which is more efficient.

Overall, we believe that BISM can be used as an alternative to AspectJ for lightweight and expressive runtime verification and even runtime enforcement (cf. [8,11,12]) thanks to its ability to insert bytecode instructions.

References

1. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5
2. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. Int. J. Softw. Tools Technol. Transf. **21**(1), 31–70 (2019)
3. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci and Falcone [1]. https://doi.org/10.1007/978-3-319-75632-5
4. Binder, W., Moret, P., Tanter, É., Ansaloni, D.: Polymorphic bytecode instrumentation. Softw. Pract. Exp. **46**(10), 1351–1380 (2016)
5. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Tarr, P.L., Cook, W.R. (eds.) Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA. pp. 169–190. ACM (2006)
6. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: Adaptable and extensible component systems (2002), https://asm.ow2.io
7. Coady, Y., Kiczales, G., Feeley, M.J., Smolyn, G.: Using AspectC to improve the modularity of path-specific customization in operating system code. In: Tjoa, A.M., Gruhn, V. (eds.) Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001. pp. 88–98. ACM (2001). https://doi.org/10.1145/503209.503223
8. Falcone, Y.: You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 89–105. Springer (2010). https://doi.org/10.1007/978-3-642-16612-9_9
9. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013). https://doi.org/10.3233/978-1-61499-207-3-141
10. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 241–262. Springer (2018)
11. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci and Falcone [1], pp. 103–134. https://doi.org/10.1007/978-3-319-75632-5_4
12. Falcone, Y., Pinisetty, S.: On the runtime enforcement of timed properties. In: Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings. pp. 48–69 (2019). https://doi.org/10.1007/978-3-030-32079-9_4
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting started with AspectJ. Commun. ACM **44**(10), 59–65 (2001)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP'97. LNCS, vol. 1241, pp. 220–242. Springer (1997)

15. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
16. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Hirschfeld, R., Tanter, É., Sullivan, K.J., Gabriel, R.P. (eds.) Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD, Potsdam, Germany. pp. 239–250. ACM (2012)
17. Soueidi, C., Kassem, A., Falcone, Y.: BISM: Bytecode-Level Instrumentation for Software Monitoring, https://gitlab.inria.fr/monitoring/bism-tool
18. Spinczyk, O., Lohmann, D., Urban, M.: AspectC++: An AOP extension for C. Software Developer's Journal (01 2005)
19. The Apache Software Foundation: Apache commons. https://commons.apache.org, accessed: 2020-06-18